

Hierarchical Design Methodology Guide

UG748 (v 12.1) May 3, 2010





Xilinx is disclosing this Document and Intellectual Property (hereinafter "the Design") to you for use in the development of designs to operate on, or interface with Xilinx FPGAs. Except as stated herein, none of the Design may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Any unauthorized use of the Design may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

Xilinx does not assume any liability arising out of the application or use of the Design; nor does Xilinx convey any license under its patents, copyrights, or any rights of others. You are responsible for obtaining any rights you may require for your use or implementation of the Design. Xilinx reserves the right to make changes, at any time, to the Design as deemed desirable in the sole discretion of Xilinx. Xilinx assumes no obligation to correct any errors contained herein or to advise you of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or technical support or assistance provided to you in connection with the Design.

THE DESIGN IS PROVIDED "AS IS" WITH ALL FAULTS, AND THE ENTIRE RISK AS TO ITS FUNCTION AND IMPLEMENTATION IS WITH YOU. YOU ACKNOWLEDGE AND AGREE THAT YOU HAVE NOT RELIED ON ANY ORAL OR WRITTEN INFORMATION OR ADVICE, WHETHER GIVEN BY XILINX, OR ITS AGENTS OR EMPLOYEES. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DESIGN, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT OF THIRD-PARTY RIGHTS.

IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOST DATA AND LOST PROFITS, ARISING FROM OR RELATING TO YOUR USE OF THE DESIGN, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE TOTAL CUMULATIVE LIABILITY OF XILINX IN CONNECTION WITH YOUR USE OF THE DESIGN, WHETHER IN CONTRACT OR TORT OR OTHERWISE, WILL IN NO EVENT EXCEED THE AMOUNT OF FEES PAID BY YOU TO XILINX HEREUNDER FOR USE OF THE DESIGN. YOU ACKNOWLEDGE THAT THE FEES, IF ANY, REFLECT THE ALLOCATION OF RISK SET FORTH IN THIS AGREEMENT AND THAT XILINX WOULD NOT MAKE AVAILABLE THE DESIGN TO YOU WITHOUT THESE LIMITATIONS OF LIABILITY.

The Design is not designed or intended for use in the development of on-line control equipment in hazardous environments requiring fail-safe controls, such as in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support, or weapons systems ("High-Risk Applications") Xilinx specifically disclaims any express or implied warranties of fitness for such High-Risk Applications. You represent that use of the Design in such High-Risk Applications is fully at your risk.

© 2010 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

Demo Design License

© 2010 Xilinx, Inc.

This Design is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Library General Public License along with this design file; if not, see:

<http://www.gnu.org/licenses/>

The PlanAhead™ software source code includes the source code for the following programs:

Centerpoint XML

The initial developer of the original code is CenterPoint – Connective Software

Software Engineering GmbH. portions created by CenterPoint – Connective Software

Software Engineering GmbH. are Copyright© 1998-2000 CenterPoint - Connective Software Engineering GmbH. All Rights Reserved. Source code for CenterPoint is available at <http://www.cpointc.com/XML/>

NLView Schematic Engine

Copyright© Concept Engineering.

Static Timing Engine by Parallax Software Inc.

Copyright© Parallax Software Inc.

Java Two Standard Edition

Includes portions of software from RSA Security, Inc. and some portions licensed from IBM are available at <http://oss.software.ibm.com/icu4j/>

Powered By JIDE – <http://www.jidesoft.com>

The BSD License for the JGoodies Looks

Copyright© 2001-2010 JGoodies Karsten Lentzsch. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of JGoodies Karsten Lentzsch nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Free IP Core License

This is the Entire License for all of our Free IP Cores.
Copyright (C) 2000-2003, ASICS World Services, LTD. AUTHORS

All rights reserved.

Redistribution and use in source, netlist, binary and silicon forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of ASICS World Services, the Authors and/or the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

About this Manual	7
Introduction to Hierarchical Design.....	7
Hierarchical Flows	8
Additional Resources.....	8
Introduction to Partitions	9
Partition States	10
Partition Preservation Levels.....	10
Import Location	11
Deciding When to Use Partitions.....	11
The Costs and Benefits of Using Partitions.....	11
Design Considerations	13
Optimization Limitations.....	13
No Optimization Across Partition Boundaries	13
Evaluation of Constants on Partition Inputs.....	14
No Optimization of Unconnected Partition Outputs	14
Logic From One Partition Cannot be Packed With Logic From Another Partition.....	14
Architecting the Design.....	14
HDL Considerations	15
Registering Input and Output Ports	15
Managing Nets Inside and Outside a Partition.....	15
Managing High Fanout Nets.....	15
Avoiding Constants as Nets	15
Avoiding Unconnected Ports.....	16
Floorplanning Partitions.....	16
Design Preservation	16
Design Preservation Flows	17
ISE Command Line Flow	17
Creating a PXML File	17
Running Implementation.....	21
Exporting Partitions	23
Updating Partition State to Import.....	23
Iterative Design.....	23
Using SmartXplorer.....	24
Removing a Partition.....	24
Synthesis Flows.....	24
Using Synplify Pro/Premier.....	25
Using Precision	26
Using XST.....	27
PlanAhead Flow	28
Creating a New Project.....	28
Creating Partitions	29

Floorplanning Partitions	29
Implementing a Partitioned Design	30
Promoting Partitions	31
Managing Partition States	32
Managing Design Runs	34
Debugging Partitions.....	35
Implementation Errors	35
Error When Trying to Import	35
What To Do If a Design Does Not Meet Timing When Importing	35
What To Do If a Design Does Not Place	35
What To Do If a Design Does Not Route	36
What To Do If Slice Utilization Goes Up With Partitions	37
BitGen DRC Errors	37
ChipScope Support	38

About this Manual

This guide focuses on the methodologies available for design preservation across the Xilinx[®] software. Design preservation is one of the Hierarchical Design flows, as described in the section below.

This document covers the following chapters:

- Chapter 1, “Introduction to Partitions” provides an overview of partitions, the building blocks for all Xilinx software HD flows.
- Chapter 2, “Design Considerations” describes many factors to consider when deciding to use partitions and an HD flow, such as the logical and physical layout of the design, HDL coding guidelines, and whether or not to floorplan.
- Chapter 3, “Design Preservation Flows” describes how to use design preservation for both the RTL and netlist level flows using the ISE[®] command line tools and the PlanAhead[™] software graphical user interface (GUI)

This document contains the following appendix:

- Appendix A, “Debugging Partitions” provides debugging or troubleshooting techniques for errors encountered during design preservation. Information about known issues and solutions is available in Answer Record 35019; see link in the “Additional Resources” section below.

Note: Readers of this guide should be familiar with the PlanAhead software and basic functionality. For more information about the PlanAhead software, refer to:

- *PlanAhead User Guide* (UG632):
http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/PlanAhead_UserGuide.pdf
- PlanAhead tutorials:
http://www.xilinx.com/support/documentation/dt_planahead_planahead12-1_tutorials.htm

Introduction to Hierarchical Design

Hierarchical Design (HD) is an approach to designing that leverages the natural logical hierarchy of a design to overcome the restrictions imposed by a typical flat design flow. While a flat flow has the advantage of being able to optimize the entire design at once, HD enables features not available in a flat flow. HD addresses the problems of long and numerous synthesis and implementation runs, difficulty meeting and maintaining timing goals, and inconsistent implementation results. HD enables the designer to break up the design into smaller, logical blocks, allowing each major function to be worked on independently. These capabilities are new to FPGA designing and they open up several new flows to FPGA designers.

Hierarchical Flows

Current software support of Hierarchical Design flows consists of:

- **Design Preservation** – A complex design is partitioned into smaller pieces or modules, and once individual modules are finalized and timing has been met, results are preserved.
- **Partial Reconfiguration** – User-defined sections of the FPGA can be reconfigured on the fly, allowing for logic changes, power savings, and resource savings, without affecting the rest of the design.
- **SCC**– Single Chip Cryptography (SCC) enables the user to address strict security and reliability requirements of a design into a single FPGA rather than in multiple FPGAs as it has been done in the past. This is achieved using a flow that leverages the Partial Reconfiguration solution, as well as Design Preservation in some cases.

Hierarchical Design flows that are coming soon to the Xilinx ISE Design Suite software include:

- **Team Design** – This is a method for breaking up the coding, implementation, and verification of a design among several team members. The final design is completed by importing each team member's block into a final implementation in which placement and routing of each block is maintained.
- **IP Reuse** – In Intellectual Property (IP) Reuse a portion of Xilinx IP, third-party IP, or user IP can be imported into a design using the previously verified placement and routing results of the IP. This eliminates the need to implement, verify timing, or perform functional verification on the IP.

Additional Resources

For more information about Partial Reconfiguration, refer to the *Partial Reconfiguration User Guide* (UG702) available at <http://www.xilinx.com/tools/partial-reconfiguration>.

For more information on the SCC flow, visit the SCC site at <http://www.xilinx.com/member/crypto/index.htm>. This secure site requires registration to access the information and documentation.

For information about the PlanAhead software, please see the following documents:

- *PlanAhead User Guide* (UG632):
http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/PlanAhead_UserGuide.pdf
- PlanAhead tutorials:
http://www.xilinx.com/support/documentation/dt_planahead_planahead12-1_tutorials.htm

For general PlanAhead information, online demos, and white papers, go to:
<http://www.xilinx.com/planahead>.

For information about known issues and solutions, see Answer Record 35019:
<http://www.xilinx.com/support/answers/35019.htm>.

Introduction to Partitions

This chapter contains the following sections:

- Partition States
- Partition Preservation Levels
- Import Location
- Deciding When to Use Partitions
- The Costs and Benefits of Using Partitions

In all Hierarchical Design (HD) flows, a design is broken up into blocks. These blocks are referred to as *partitions*, and they are the building blocks for all Xilinx® software HD flows. They are used to define hierarchical boundaries so that a complex design can be broken up into smaller, more manageable pieces. Partitions create boundaries, or insulation, around the hierarchical module instances, isolating them from other parts of the design. A partition that has been implemented and exported can be re-inserted into the design using a simple cut-and-paste type function, which preserves the placement and routing results of the module instance.

All of the partition definitions and controls are specified in the `xpartition.pxml` (PXML) file. This file is read in when the tools are run. The PXML file can be created by hand, with or without the use of the available PXML template, or through the PlanAhead™ software graphical user interface (GUI). See Chapter 3, “Design Preservation Flows” for information on XPML file creation.

In the 12.x release, the following FPGA architectures support partitions:

- Spartan®-3 platform
- Spartan-3E platform
- Spartan-3L platform
- Spartan-3A platform
- Spartan-3A DSP platform
- Spartan-6 family
- Virtex®-4 family
- Virtex-5 family
- Virtex-6 family

Partition States

A partition can be implemented or imported depending on the current state of the partition. The first time a partition is run through the ISE Design Suite implementation tools, such as NGDBuild, Map and PAR, the state of the partition must be set to *implement*. After implementation completes, a partition can be exported so that the results can be re-imported for a future run. However, the exported results are only valid for future implementations if the internal logic and the partition interface have not changed.

When a change is made to a partitioned module, the placement and routing information for that partition becomes out of date. The modified partition must be re-implemented, while unchanged partitions can be imported from a previous run. Changes to the partition that results in partition information going out of date include:

- Changes to the HDL code, or anything that changes the netlist associated with a partition.
- Changes to the physical constraints associated with a partition, including AREA_GROUP and LOC constraints.
- Changes to the target architecture, including device, package, and speed grade.
- Adding or changing connections on a ChipScope™ analyzer core that is connected to the partition.

If an exported partition becomes out of date, set the State attribute in the PXML file to correctly manage the state of the partition. Failure to do so results in errors from the implementation tools.

Changes to a partition that do not force a re-implementation of the partition include:

- Constraint changes that do not affect the physical location of logic (i.e., TIMESPEC)
- Changes to implementation options that differ from those used by the original partition (i.e., `par -xe`)

Partition Preservation Levels

The main purpose of the partition is to preserve the results of a run by importing previous results. The level of preservation can be specified. By default, 100% of placement and routing is preserved when importing a partition.

The default behavior can be modified to preserve placement results (no routing), or synthesis results (no placement or routing). However, it is important to note that regardless of the preservation level, the entire partition is imported with a copy of all implementation information including placement and routing. The preservation level then determines the amount of flexibility that the implementation tools have to modify the imported placement and routing to improve results. Relaxing preservation levels on some partitions can free up device resources, giving the tools more flexibility to place and route other partitions. The preservation level can be set per partition and can apply to imported partitions only.

If a timing critical partitioned module has met timing and no changes are expected (for example, an IP core), the routing preservation level is a good choice. If a partition is not timing critical or has not met timing yet, relaxing the preservation level will give the tools more flexibility for finding a solution.

If the goal of using partitions is to reduce verification time, then the preservation level should always be routing. If the preservation level needs to be changed in order to meet timing or finish routing another part of the design, the partition should be re-verified.

Floorplanning partitions may reduce the need to relax the preservation level.

Import Location

When importing a partition, the location of the exported results must be specified. When an implemented design is exported, every partition in the design is automatically exported with it. Xilinx recommends using a single export directory from which to import, whenever possible. When a partition is imported to a design, the associated design is opened in memory. If you import partitions from multiple design runs (multiple locations), every design is opened in memory. This increases the total memory used and the total runtime of the implementation tools. This increases the likelihood of a routing collision also, which would require some nets to be rerouted.

Deciding When to Use Partitions

Partitions should only be used on modules that need them. Overuse of partitions can have negative effects on runtime and performance. For modules that are not an isolated functional block in their own hierarchy or that would benefit from global optimization with other blocks, a flat optimization will provide the best results.

In order to ensure success when using partitions, adhere to the design consideration in Chapter 2, “Design Considerations.”

Good candidates for using partitions may be the following:

- A functional block, such as a DSP module or an EDK system.
- High performance cores.
- Instances containing logic, which need to be packed or placed together within the device.
- Modules that adhere to good design practices.

The Costs and Benefits of Using Partitions

While there are several benefits of using an HD flow, there are costs associated with partitions as well. The primary impact from using partitions is that the hierarchical boundary that is formed affects optimization. No optimization can be done across a partition boundary, and, if a design does not account for this, adding partitions can have a significant impact on timing, utilization, and runtime. Even with careful planning there are other optimization and packing limitations that can increase utilization and can have negative effects on timing. These effects should be minimal for well architected designs but are worth noting.

For more information on how partitions affect optimization and how to design to minimize these effects, refer to the next chapter.

Design Considerations

The decision to use a Hierarchical Design (HD) flow is made at the beginning of the design planning phase and not once a design is experiencing problems with timing closure or inconsistent results. In order to get the full benefit of an HD flow, there are many factors that need to be considered up front. These include the logical and physical layout of the design, HDL coding guidelines, and the use of floorplanning constraints. These topics are discussed in more detail below.

This chapter contains the following sections:

- Optimization Limitations
- Architecting the Design
- HDL Considerations
- Floorplanning Partitions

Optimization Limitations

When using partitions in a design, the optimization limitations must be considered. Below is a list of optimization limitations inherent to the insulation created by partitions. It is important to remember that when even a single partition is added to a design, every instance in the design becomes part of a partition. Instances not specified as their own partition become part of the top partition. Below are some optimization limitations and examples of how each can affect the logic in the design. Methods for avoiding or minimizing the effects of these limitations are discussed also.

No Optimization Across Partition Boundaries

This limitation includes optimization between a parent and child partition, as well as between two child partitions. This can affect timing and utilization when compared to a flat design.

If combinatorial logic that drives more combinatorial logic has a partition boundary in between the logic, no optimizations can occur on this logic.

If partitions share common logic, resource sharing will not occur. If a control bus that goes to four different partitions has identical decode logic in all four, then that logic will be replicated four times. In a flat flow, this logic can be shared. Similarly, if the bus is registered in multiple partitions, multiple sets of registers will get inferred. In a flat flow, these could be optimized down to a single set of registers and shared.

Evaluation of Constants on Partition Inputs

If inputs to a partition are tied to a constant value in order to guide optimization, this constant cannot be pushed across the partition boundary and optimization will not occur.

This can occur when a constant is used to enable or disable a specific feature in a core/module. Xilinx does not recommend controlling logic in a module via ports, which will not work with partitions. However, we do recommend using parameters/attributes or including a package file instead.

No Optimization of Unconnected Partition Outputs

If an output of a partition is not driving anything, the source logic will not be optimized as it would in a flat flow.

Logic From One Partition Cannot be Packed With Logic From Another Partition

This can affect utilization if the FF-to-LUT ratio highly favors one or the other.

If combinatorial logic inside of a partition drives an output that is eventually an FF, the LUT cannot be packed with an FF.

Architecting the Design

When implementing a design using a flat flow, synthesis and implementation tools can optimize the design for speed and area as a whole. Since the tools can optimize across hierarchical boundaries, the logical layout of the design is not as critical. When implementing a design in an HD flow, partitions act as an optimization barrier in order to isolate the logic. This can have dramatic affects on the design. Take the design in Figure 1 as an example.

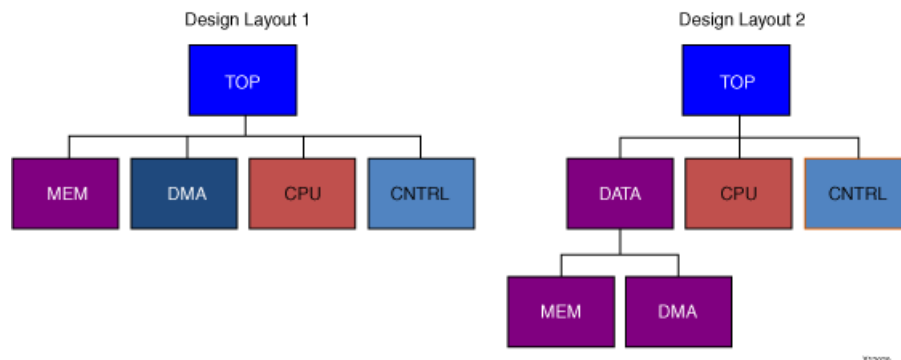


Figure 1: Sample Design Hierarchy

The design layout on the left shows the modules “MEM” and “DMA” at the same hierarchical level. If partitions were added to all of these modules under “TOP”, optimization between “MEM” and “DMA” could not occur. If these two modules have a lot of related logic that would benefit from the optimization done in a flat flow, they will not get this optimization in the example on the left. This could lead to increased utilization and poor timing results. However, if the design hierarchy is modified to match the layout on the right, modules with shared logic can be grouped together under one partition, “DATA”. The same optimization for “MEM” and “DMA” that was not possible in flat flow can be achieved in an HD flow by using a partition.

HDL Considerations

Using an HD flow is something that should be considered prior to architecting the design, defining module interfaces and coding the modules. These flows are not timing closure techniques to be used on a design after timing issues arise in a flat flow. There are several coding guidelines and recommendations that will enable a design to achieve the intended benefits of an HD flow.

The partition creates insulation for the module that limits the ability of the software to optimize across the boundary. To avoid issues and improve performance when using partitions, the following guidelines should be followed.

- Register input and output ports
- Manage nets used to drive logic inside and outside of the partition
- Manage high fanout nets
- Do not use constants as inputs to a partition
- Do not leave input or output ports unconnected

Registering Input and Output Ports

It is very important to register inputs and outputs, if possible. Since no optimization occurs across the partition boundary, inputs and outputs of partitions can quickly become timing bottlenecks. Registering the inputs and outputs enables the tools to focus on paths that are internal to the partition, which allows for true timing preservation on the module. Nets that cross a partition boundary are not preserved unless all partitions connected to the net are imported. As a result, timing critical nets that cross this boundary may continue to cause violations in the partition even when imported. If the boundary nets are registered, this will prevent any timing critical changes internal to the partition from occurring.

Managing Nets Inside and Outside a Partition

It is important not to use a net as both an internal and output net within the partition. If nets need to be used in the partition and as output ports, duplicate the source of the nets. Then, use one net internally and the other as the output port.

Managing High Fanout Nets

Fanout on outputs of partitioned modules must be taken into account. If an output of a partition has a high fanout and connects to multiple areas in the design, the driver may need to be replicated. In the flat flow, this replication occurs automatically, but in an HD flow, you must replicate the driver by hand.

Avoiding Constants as Nets

Some netlist modules, like IP cores, have been designed with the assumption that Map will optimize away unneeded portions of logic. This is generally achieved by tying inputs of the core to constants that either enable or disable certain logic. This enables IP to be delivered as a netlist while still allowing some customization. However, this method does not work well with partitions because of the optimization boundary. If a partition is added directly to an EDIF/NGC core for which Map is expected to optimize logic based on constants on the ports, no logic optimization will be done and the results will not be ideal. If you deem it necessary to put a partition on a core that exhibits this behavior, an HDL wrapper should be added to wrap the EDIF/NGC core, and the partition placed on the wrapper. The port list of the HDL wrapper should only contain the necessary IO connecting the IP core to the rest of the design, and all

constant assignments can remain in the wrapper logic. With the partition defined a level above the IP core, Map is free to trace the constant into the core and do any necessary optimization.

Avoiding Unconnected Ports

Unconnected outputs have a similar effect on optimization. If an output of a partition is unconnected, Map cannot optimize the driver associated with this source-less net like it can in a flat design. If this logic cannot be optimized in the HDL code, adding a wrapper around the partition will move the partition boundary and allow Map to optimize the logic.

Similarly, an output of a partition that is not driven by logic and that connects to logic outside of the partition can cause errors. In this case, the implementation tools cannot see that the partition output is not driven, and the tools will route a partial net. This is not valid and will cause DRC errors in BitGen. You must remove any unnecessary partition ports from the code. The partition will then need to be re-implemented.

Floorplanning Partitions

Floorplanning refers to controlling the placement of a design through constraints, and this section specifically refers to using AREA_GROUP constraints.

There are no restrictions with AREA_GROUP constraints on partitioned designs; however, Xilinx recommends creating slice ranges on a CLB boundary. This will maximize the available resources for placement and routing. A quick way to verify that a slice range was correctly aligned to a CLB boundary is to check the XY coordinates of the constraint. The range is on a CLB boundary if the XY starts on an even number and ends on an odd number, such as X0Y0 to X3Y9. You can use the PlanAhead™ software to create the AREA_GROUP constraints and to snap the constraints to CLB boundaries for you also. For more information on CLBs or other blocks in the FPGA, refer to the device-specific data sheet: http://www.xilinx.com/support/documentation/data_sheets.htm.

Floorplanning has an additional benefit with partitions by keeping all the logic associated with a partition in one area of the device. This will create a region for placing and routing each partition, which will minimize the risk of routing conflicts during the import process. This is also useful for reserving other parts of the FPGA for additional logic that will be designed in later.

Although partitions are supported in the PlanAhead tools, you can run the partitioned design using the command line tools also. In this case, PlanAhead can be used to set up the partitions, floorplan the design, and create the `xpartition.pxm1` file for use in the ISE® command line flow. See Chapter 3, “Design Preservation Flows” for more information on supported flows.

Design Preservation

Design preservation partitions do not require an AREA_GROUP constraint. However, for some designs, floorplanning can improve runtime and timing results. Using the AREA_GROUP constraint can also reduce the chance of potential routing conflicts in the import process.

Design Preservation Flows

The goal of design preservation is to reduce the number of implementation iterations during the timing closure phase. Once you meet timing on a portion of the design, the implementation results (placement and routing) are used in the next iteration. This prevents portions of the design that are complete and that meet timing from being affected by design changes in other parts of the design.

Design preservation uses partitions to keep the previous implementation results for a module instance. Careful use of partitions reduces the timing closure phase by reducing the number of design iterations, and eliminates re-verification of unchanged instances. Preserving portions of the design can affect the implementation runtime, depending on which modules are being implemented and which modules are being preserved. If the implemented module has very tight timing requirements, runtime can be longer. If the implemented module easily meets timing and the critical paths are in the preserved modules, runtime can be shorter.

This chapter describes how to use design preservation for both the RTL and netlist level flows using the ISE® Design Suite command line tools or the PlanAhead™ software graphical user interface (GUI). This chapter contains the following sections:

- ISE Command Line Flow
- Synthesis Flows
- PlanAhead Flow

ISE Command Line Flow

The ISE Design Suite command line flow uses partitions on a post-synthesis design. This design flow is command-line driven and uses the Xilinx® NGDBuild, Map and PAR implementation software tools.

When the implementation tools run, they look for the `xpartition.pxml` file in the current working directory automatically. The current working directory is the directory from where NGDBuild, Map, and PAR are being run. The implementation tools use this file to check where partitions are defined in the design and what the partition states are.

Creating a PXML File

All partition definitions are contained with the `xpartition.pxml` file. The PXML file is case-sensitive, and must be named `xpartition.pxml`. The top-level module of the design must be defined as a partition in addition to any optional lower-level partitions. Child or nested partitions are supported.

The PXML file can be created by hand in a text editor or in a software tool, such as the PlanAhead software graphical user interface (GUI). When located in the current working directory, the PXML is automatically recognized by the implementation tools.

A template for the `xpartition.pxml` file is available for creating a PXML file by hand. The file is located in the install directory at:

```
<Xilinx_12_directory>/PlanAhead/testcases/xpartition.pxml
```

The following is a sample PXML file.

```
<?xml version="1.0" encoding="UTF-8" ?>

<Project FileVersion="1.2" Name="Example" ProjectVersion="2.0">
  <Partition Name="/top" State="implement" ImportLocation="NONE">
    <Partition Name="/top/module_A" State="import"
    ImportLocation="/home/user/Example/export" Preserve="routing">
      </Partition>
    <Partition Name="/top/module_B" State="import"
    ImportLocation="../export" Preserve="routing">
      </Partition>
    <Partition Name="/top/module_C" State="implement"
    ImportLocation="../export" Preserve="placement">
      </Partition>
    </Partition>
  </Project>
```

Figure 2: Sample xpartition.pxml file

The information below defines the attributes and values that are used in the sample PXML file above.

Table 1: PXML attributes for *Project* definition

Attribute name	Value	Description
FileVersion	1.2	Used for internal purposes. Do <i>not</i> change this value
Name	Project_Name	Project_Name is user defined.
ProjectVersion	2.0	Used for internal purposes. Do <i>not</i> change this value

Table 2: PXML attributes for *Partition* definition

Attribute name	Value	Description
Name	Partition_Name	Hierarchical instance name of module where the partition should be applied.
State	implement	Partition will be implemented from scratch.
	import	Partition will be imported and

		preserved according to the level set by Preserve.
ImportLocation	Path	Ignored if State does not equal "import". Path can be relative or absolute, but the location specified must contain a valid "export" directory when "State=import". "NONE" is a predefined keyword for no import directory.
Preserve	routing	100% placement and routing is preserved. This is the default for top-level partition.
	placement	Placement is preserved, and routing can be modified.
	synthesis	Placement and routing can be modified.
	inherit	Inherit value from the parent partition. This is the default for all partitions except the top-level partition.

If you create the PXML file by hand or from a third-party party synthesis tool, skip to the “Running Implementation” section later in this chapter.

To create an `xpartition.pxml` file using PlanAhead, follow the steps below:

1. In the PlanAhead software, create a new netlist project and point to the top-level netlist, and any lower-level netlists that define the partitions.
2. Click the netlist design to load it.
3. Open the Netlist tab, and select the module instances to be marked as partitions.
4. From the right-click menu, select **Set Partition**, as shown in Figure 3.

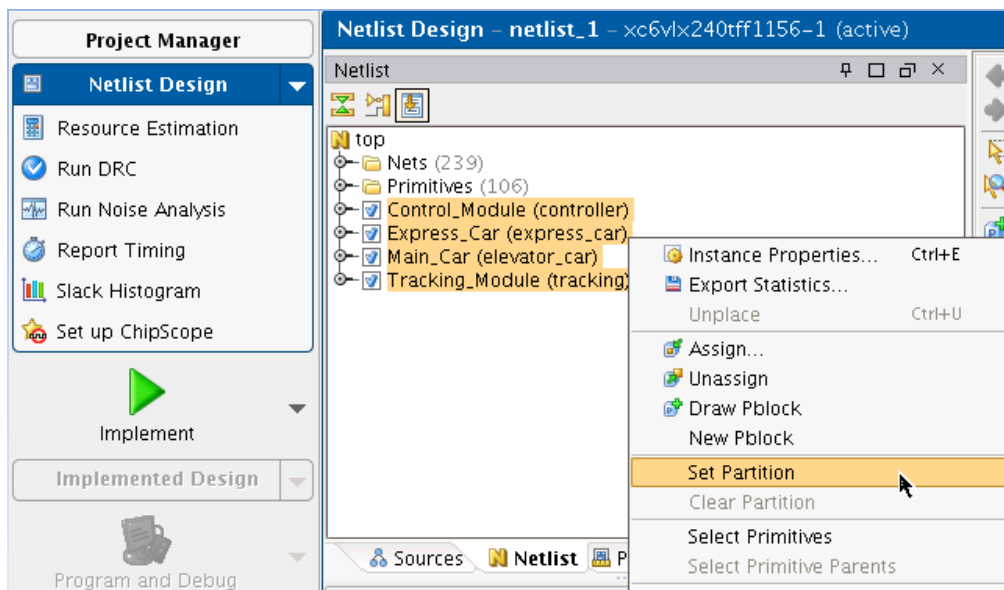


Figure 3: Setting Partitions in PlanAhead

- From the Implement button pull-down, select **Implementation Settings**, as shown in Figure 4.

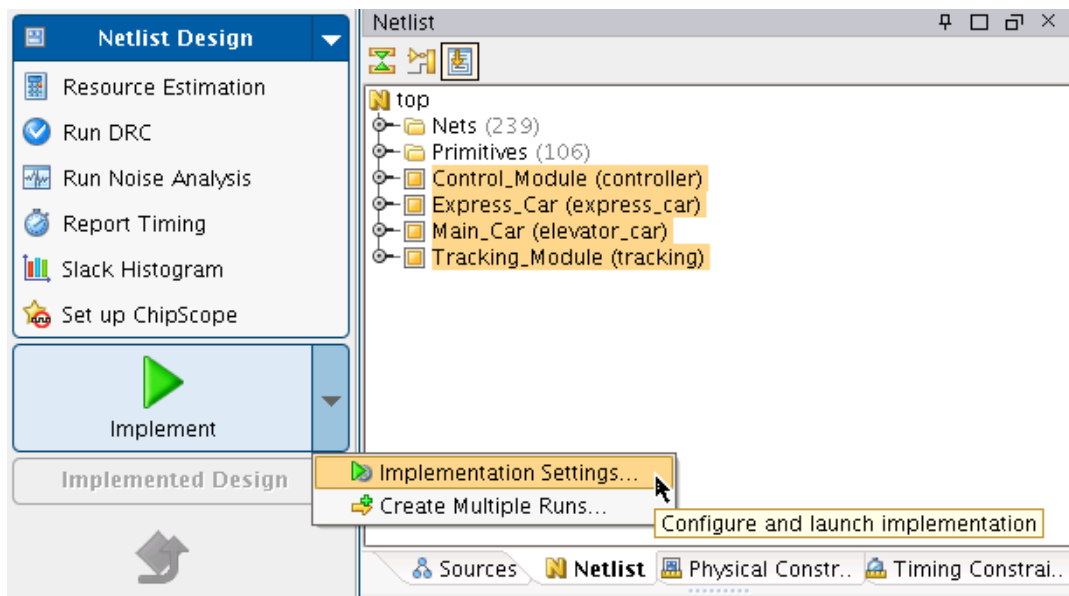


Figure 4: Implementation Settings

- In the Implementation Setting dialog box, open the Launch Options dialog box by clicking on the button next to current launch options, as shown in Figure 5.

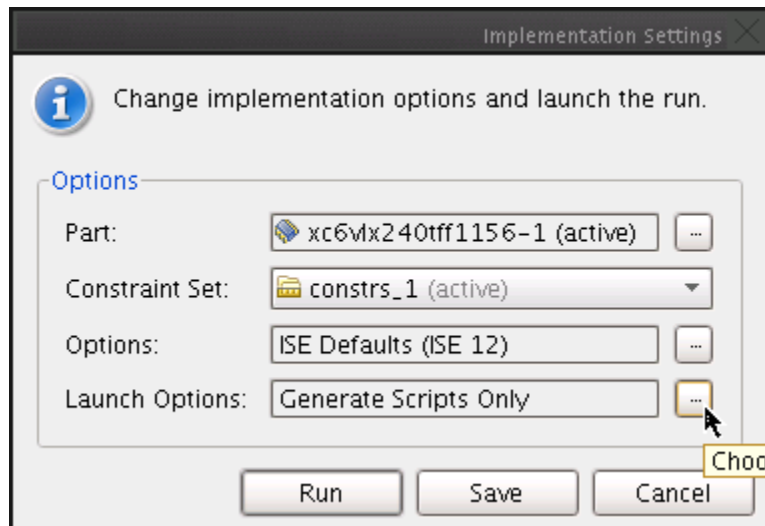


Figure 5: Launch Options

7. In the Launch Options dialog box, select the **Generate scripts only** radio button and click **OK**.
8. Click **Run**.

The steps to run the Design Preservation flow from PlanAhead are described in the "PlanAhead Flow" section later in this chapter.

For more details instructions on how to use PlanAhead, see the *PlanAhead User Guide* (UG632): http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/PlanAhead_UserGuide.pdf.

The `xpartition.pxml` file is generated in the `<project_name>.runs/impl_1` directory. This file can be copied to a location where the ISE command line flow will be run.

Running Implementation

Once the `xpartition.pxml` file is created, the Xilinx implementation tools can be run in command line or batch mode. A basic script might look like:

```
ngdbuild -uc design.ucf design.edn design.ngd
map -w design.ngd -o design_map.ncd design.pcf
par -w design_map.ncd design.ncd design.pcf
```

Note: The netlist specified in the NGDBuild command must *not* be the output of a flat synthesis run. The synthesis project must have partitions in mind when being set up and run. For more information on synthesis flows, see the "Synthesis Flows" section later in this chapter.

To verify that the `xpartition.pxml` file has been recognized by the tools and that the states of the partitions are set correctly, you can look at the report files for the various implementation tools. For example, if you look at the NGDBuild report file (`.bld`), you will see a section near the bottom with the partition implementation status, as shown in Figure 6.

```
Partition Implementation Status
-----

Preserved Partitions:

Implemented Partitions:

Partition "/top":
Attribute STATE set to IMPLEMENT.

Partition "/top/Control_Module":
Attribute STATE set to IMPLEMENT.

Partition "/top/Express_Car":
Attribute STATE set to IMPLEMENT.

Partition "/top/Main_Car":
Attribute STATE set to IMPLEMENT.

Partition "/top/Tracking_Module":
Attribute STATE set to IMPLEMENT.
```

Figure 6: Sample NGDBuild Report with Partition Status

The Map and PAR report files have similar partitions sections to the NGDBuild report.

Implementation Options Not Supported With Partitions

Partitions do not support the following implementation options.

- Global Optimization (**-global_opt**)
- SmartGuide™ technology (**-smartguide**)

Note: There are many Xilinx environment variables (XIL_*) that have a wide range of effects on the implementation tools, and the effects may vary from one ISE release to the next. Design preservation has not been tested with environment variables, so remove all special Xilinx environment variables prior to running the tools.

Exporting Partitions

When the implementation results are satisfactory, the partitions should be exported. To export partitions simply copy over all of the files in the implementation to an export directory. If an exact copy of the implementation directory is too large, a script could be used to copy just the required files along with the suggested important files listed below.

Required Files

The required files are the `*prev*.*` files and the `xpartition.pxml` file. The `*prev*.*` files contain the previous implementation data required to import a partition. The PXML file must be included in order to verify that it is appropriate to import a partition. For example, if the PXML file doesn't contain the partition that is being imported, then an error will be generated.

Optional Files

The report files from NGDBuild (`.bld`), Map (`.mrp` and `.map`), PAR (`.par`), TRCE (`.twr/ .twx`) and the UCF are optional but important files. These files document what options were run when the design was implemented.

It is very important not to modify the files in the exported directory in any way. They become source-like files.

If there are multiple partitions in a design, they are all exported. It is easier to think of the entire design as being exported than a specific partition. For faster runtime, it is a best practice to have one directory with all exported data. This limits the number of design files that have to be loaded into memory when partitions are imported.

Updating Partition State to Import

Once the partition has been exported, the `xpartition.pxml` file needs to be updated to reflect the current state of the partitions. This can be done with a text editor. If the desired result is to import a partition that has been exported, the partition state should be set to 'import'. The `ImportLocation` attribute should be set to the location of the exported partitions. You can specify a relative path (such as, `../export`), or an absolute path (such as, `/home/user/Example/export`).

Iterative Design

Make any design changes as necessary and re-synthesize. If synthesis modifies a partition that has been exported, you must manually change the partition state for the corresponding partition from 'import' to 'implement' in the `xpartition.pxml` file. You will get an error in NGDBuild if you try to import a partition that does not match exactly the partition from the import location.

```
ERROR:NgdBuild:1037 - The logic for imported partition '/top/Main_Car' using
previous implementation './export/top_prev_built.ngd' has been modified.
```

This situation can occur when the source for the partition was modified and the partition was not re-implemented and exported. The partition must be re-implemented and exported before it can be imported into the design. It should be noted that adding comments to the HDL may result in small logic or naming changes in the netlist, which requires that the partition be implemented on the next iteration.

If a partition is imported with preservation level set to anything other than routing, and the placement and routing are modified by PAR, the partition must be exported so that it can be used in the next iteration.

Using SmartXplorer

SmartXplorer can be used on designs containing partitions just as it can be on a flat design flow. On completed partitioned portions of the design that have tight timing budgets, you can run SmartXplorer to find a solution that meets timing. Then, the standard implementation flow can be used to implement other portions of the design that are changing while importing the SmartXplorer results for the timing critical modules.

When importing partitions from a SmartXplorer run, the import location in the PXML file needs to point to the directories created by SmartXplorer. The path to this directory can be relative (such as, `./run2`) or absolute (such as, `/home/user/Example/run2`).

SmartXplorer can be invoked with either the top-level netlist or the NGD file as the input. If an NGD file is specified, there are a couple of considerations.

- The NGD file must be created using the same `xpartition.pxml` file that Map and PAR will use. If no PXML file was present when NGDBuild was run, then Map and PAR will ignore the PXML file.
- SmartXplorer will run Map and PAR, and the NGD file will be one level up. This means that to import the SmartXplorer results, the NGD files one level up must be copied into the appropriate “runs” directory created by SmartXplorer prior to importing from that location.

For more information on SmartXplorer, see the *Command Line Tools User Guide* (UG688): http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/devref.pdf

Removing a Partition

To remove a partition, simply remove the reference from the `xpartition.pxml` file, and set the state of the parent partition to ‘implement’. To remove all partitions and run a flat flow, rename or remove the `xpartition.pxml` file from the implementation directory. Partitions can be added again by adding the `xpartition.pxml` file back to the implementation directory. You can import the partitions, assuming that the input netlists have not been changed and the export directory still exists.

Synthesis Flows

Design preservation requires that each partition be synthesized by itself in order to prevent a design change in one area causing different synthesis results in another area. This is accomplished by using an incremental synthesis approach or a bottom-up synthesis approach.

Most third-party synthesis tools offer an incremental synthesis approach where RTL modules can be marked as partitions. When synthesis runs, each partition is synthesized separately with hard hierarchical boundaries, which prevents an HDL change in one module from affecting another module. These tools also determine which module or instance should be re-synthesized based on HDL or constraint changes.

The supported incremental synthesis flows are as follows:

- Synopsys Synplify Pro/Premier, which uses compile points.
- Mentor Precision, which uses attributes in HDL to specify partitions.

The second approach is a bottom-up synthesis approach. In this flow, each partition has a separate synthesis project and resulting netlist. You can decide which synthesis projects need to be run based on their HDL code or synthesis constraint changes. The top level is synthesized by using black boxes for the lower-level modules, and the lower-level modules are synthesized without inferring IO or clock buffers. This approach is supported by third-party synthesis tools and Xilinx Synthesis Technology (XST).

Advantages of the vendor-specific incremental synthesis flow are as follows:

- No need to create separate synthesis project files for each partition.
- An easier transition from a flat synthesis flow.
- The synthesis tool determines which modules are re-synthesized based on HDL code changes and timing constraint changes.

Advantages of a bottom-up flow are as follows:

- Multiple engineers can work on the same overall design during synthesis because each engineer has his/her own synthesis project.
- Offers complete control over which instances are re-synthesized. It is easier to determine which instances are re-synthesized because the user determines which project is re-synthesized. Each netlist has a unique time stamp.

Using Synplify Pro/Premier

There are two ways to use Synplify and Synplify Pro/Premier for design preservation: a basic bottom-up synthesis flow, and the block based or incremental synthesis flow using compile points.

Synplify Bottom-Up Flow

For the bottom-up flow, each instance will have a corresponding Synplify project file. Care must be taken to not infer IOBs or global clock buffers in the lower-level project files.

The following turns off I/O insertion in the project file:

```
set_option disable_io_insertion 1
```

The `syn_noclockbuf` attribute is used to automatically turn off clock buffer usage. This can be put in the Synplify constraints (SDC) file:

```
define_attribute { clock_port } syn_noclockbuf 0
define_global_attribute syn_noclockbuf 0
```

The `syn_noclockbuf` attribute can also be put directly into Verilog and VHDL code. See the Synplify documentation for more examples.

Synplify Pro/Premier Incremental Synthesis Flow

The incremental synthesis flow is available in Synplify Pro and Synplify Premier. This flow uses compile points to break down the design into smaller synthesis units. Use the locked mode to create a solid boundary with no logic moving in or out of the compile point. The soft and hard modes will allow optimizations across boundaries, which are not supported. Any netlist changes to a partition will require that a partition be implemented. NGDBuild issues an error if the netlist does not match the imported partition.

An example compile point in the SDC file is as follows:

```
define_compile_point {v:controller} -type {locked} -cpfile {}
```

Use Synplify to create and modify any compile points in the SDC file. The synthesis report file contains the status of each compile point (Figure 7).

```

Summary of Compile Points

Name                Status      Reason
-----
controller          Unchanged  -
elevator_car        Remapped   Design changed
express_car         Remapped   Design changed
top                 Remapped   Design changed
=====

```

Figure 7: Sample Synplify Synthesis Report with Compile Points

After synthesis, there are three choices to run the Xilinx implementation tools.

- **ISE Command Line Flow** - By running a simple Tcl command, Synplify creates an updated PXML file to use with the Xilinx implementation tools. For more information, see the “ISE Command Line Flow” section.
- **PlanAhead Flow** – This GUI flow imports the synthesis netlists, and the design partitions are redefined inside of PlanAhead. For more information, see the “PlanAhead Flow” section.
- **Synplify Cockpit** - Run the Xilinx implementation tools from within the Synplify cockpit. For more information, see the Synplify Pro/Premier product documentation available at <http://www.synopsys.com/home.aspx>.

Using Precision

The Mentor Precision synthesis tool supports the Design Preservation flow. The most common flow is the partitions-based incremental flow. In this flow, you can specify partitions using attributes. These can be set on a module or instance, as shown in Figure 8.

```

Verilog Module:
module my_block( input clk; ...) /* synthesis incr_partition */;

Verilog Instance:
my_block my_block_inst( .clk(clk), ... .data_out(data_out) ); // synthesis attribute my_block_inst
incr_partition true

VHDL Module:
entity my_block is port( clk: in std_logic; ...); attribute incr_partition : boolean; attribute incr_partition
of my_block : entity is true; end entity my_block;

VHDL Instance:
component my_block is port( ... end component; ... attribute incr_partition : boolean; attribute
incr_partition of my_block_inst : label is true; ... my_block_inst

```

Figure 8: Sample Precision Synthesis Report with Attributes

The synthesis report shows whether or not a partition is being optimized based on the partition state, as shown in Figure 9.

```
[16027]: Incremental: Skipping Optimize for <...>.fifo_16_64.rtl_unfold_0
[16027]: Incremental: Skipping Optimize for <...>.fir_filter.rtl_unfold_0
[15002]: Optimizing design <...>.fsm.rtl_unfold_0
[16027]: Incremental: Skipping Optimize for <...>.fir_top.rtl
```

Figure 9: Sample Synthesis Report

After synthesis, there are three choices to run the Xilinx implementation tools.

- **ISE Command Line Flow** – Click the Place & Route button inside of Precision to create a PXML file to use with the Xilinx implementation tools. For more information, see the “ISE Command Line Flow” section.
- **PlanAhead Flow** – This GUI flow imports the synthesis netlists, and the design partitions are redefined inside of PlanAhead. For more information see the “PlanAhead Flow” section.
- **Precision** - Run the Xilinx implementation tools from Precision using the Place & Route button. This will invoke the ISE tools automatically. For more information on this flow, refer to a more detailed application note on the Mentor Graphics SupportNet site: <http://supportnet.mentor.com/>.

Using XST

XST currently supports a bottom-up synthesis flow for design preservation.

XST Bottom-Up Flow

Each instance that is used as a partition must have its own synthesis project file, including the top partition. Be careful not to infer IOBs or global clock buffers in lower-level partitions if they are contained in the top partition.

Use the following option in the XST file to prevent IOs from being inferred.

```
-iobuf NO
```

IOBs can be used in a lower-level partition, but then steps must be made to prevent an additional IOB from being inferred in the top level. You can turn off IOB insertion for the entire top level by using the **-iobuf** option, or for individual signals by putting a **BUFFER_TYPE=NONE** attribute on the individual signal names. The **BUFFER_TYPE=NONE** attribute can also be used to prevent global buffers (BUFG) from being inferred in the top level when a lower-level partition has a BUFG instantiated.

For more information on how to set the BUFFER_TYPE attribute and how to run XST in command line mode, please refer to the *XST User Guide*:

http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/xst.pdf

PlanAhead Flow

The PlanAhead software supports partitions starting with the 12.1 software. This section gives a broad overview of the Design Preservation flow in PlanAhead. For a more detailed tutorial, please refer to the *PlanAhead Tutorial: Leveraging Design Preservation for Predictable Results* (UG747) available on the Xilinx website: http://www.xilinx.com/support/documentation/dt_planahead_planahead12-1_tutorials.htm

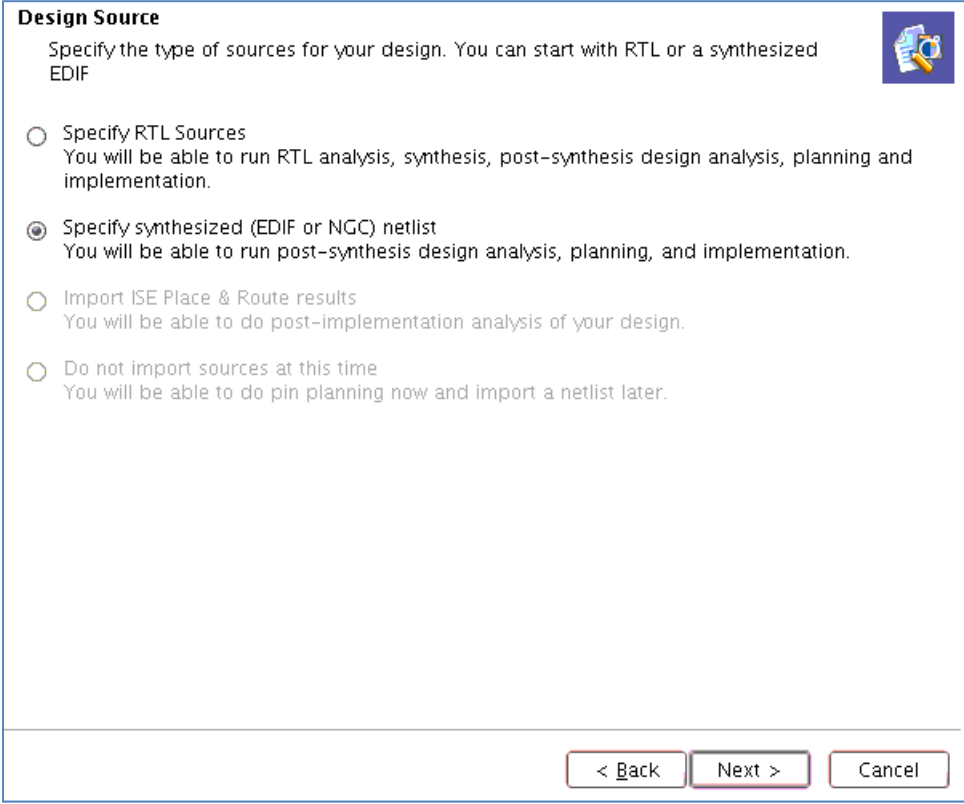
Currently, PlanAhead supports partitions in netlist projects only, which means that you cannot create an HDL-based PlanAhead project for partitions. As a result, synthesis must be run outside of the PlanAhead software. For more information on setting up synthesis projects for partitions, see the “Synthesis Flows” section.

Though not supported, it is possible to create a non-HD (flat) synthesis project in PlanAhead and add partitions to the levels of hierarchy of the project. However, since there is only a single netlist produced by synthesis, a change to one part of the design can affect portions of another.

Note: Place a partition either on a level of hierarchy that has its own independent netlist (bottom-up flow) or in a netlist created by an incremental synthesis flow. For more information on creating netlists for design preservation, see to the “Synthesis Flows” section.

Creating a New Project

When the PlanAhead software is initially launched, select the **Create New Project** option on the Getting Started page. The New Project wizard guides you through the steps of creating a new PlanAhead project. In the PlanAhead software in 12.1, you must specify a synthesized netlist as the design source. RTL support for partitions is coming in a future release of PlanAhead.



Design Source

Specify the type of sources for your design. You can start with RTL or a synthesized EDIF

- Specify RTL Sources
You will be able to run RTL analysis, synthesis, post-synthesis design analysis, planning and implementation.
- Specify synthesized (EDIF or NGC) netlist
You will be able to run post-synthesis design analysis, planning, and implementation.
- Import ISE Place & Route results
You will be able to do post-implementation analysis of your design.
- Do not import sources at this time
You will be able to do pin planning now and import a netlist later.

< Back Next > Cancel

Figure 10: Selecting Synthesized Netlist as the Design Source

Continue through the wizard and specify the top-level netlist, locations of lower-level netlists, and any existing UCF constraint files. When everything has been specified, click **Finish** to open the project in PlanAhead.

Creating Partitions

Partitions can be created in PlanAhead as follows:

1. Load the netlists into PlanAhead by selecting the **Netlist Planner** on the left side of the main window.
2. When the Netlist Planner opens, select the **Netlist** tab to view the design hierarchy.
3. Select the module instances that should be partitioned, right-click and select **Set Partition** from the popup menu (see Figure 11).

Note: Only instances that have been designed and synthesized with partitions in mind should be selected.

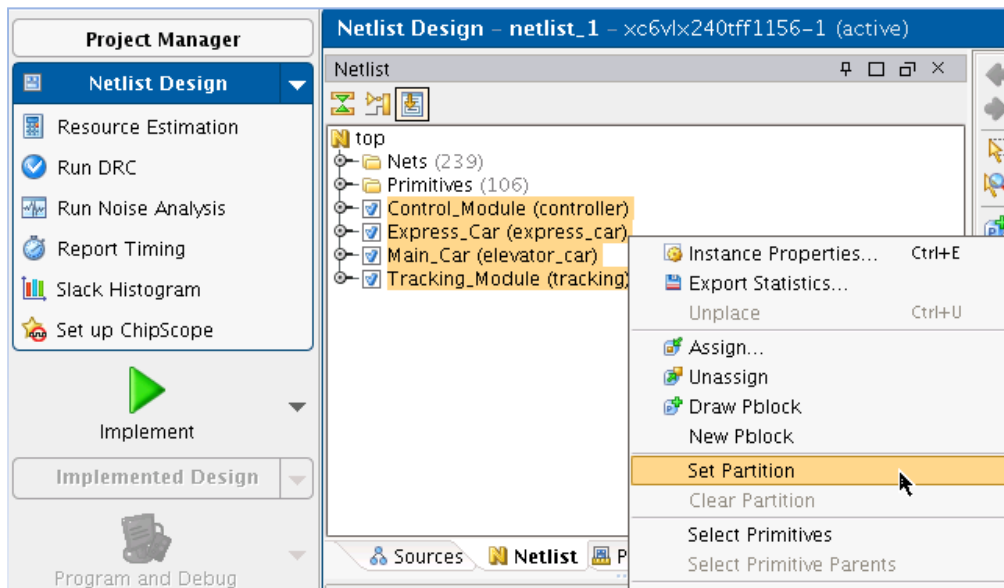


Figure 11: Setting Partitions in PlanAhead

Floorplanning Partitions

To help decide if a partition should be floorplanned, review the “Floorplanning Partitions” section of Chapter 1. In this simple example, four partitions are floorplanned to illustrate how this is done.

By selecting the partitions in the Netlist tab one at a time, a Pblock can be created for each partition. A Pblock in PlanAhead defines the AREA_GROUP constraints.

To create a Pblock:

1. Right-click on a partition netlist
2. Select **Draw Pblock**.
3. In the Device view, draw a rectangular shape around each partition.

The result may look similar to Figure 12.

For more information on how to floorplan modules in PlanAhead, see the *PlanAhead Tutorial: Design Analysis and Floorplanning for Performance* (UG676) available on the Xilinx website:

http://www.xilinx.com/support/documentation/dt_planahead_planahead12-1_tutorials.htm.

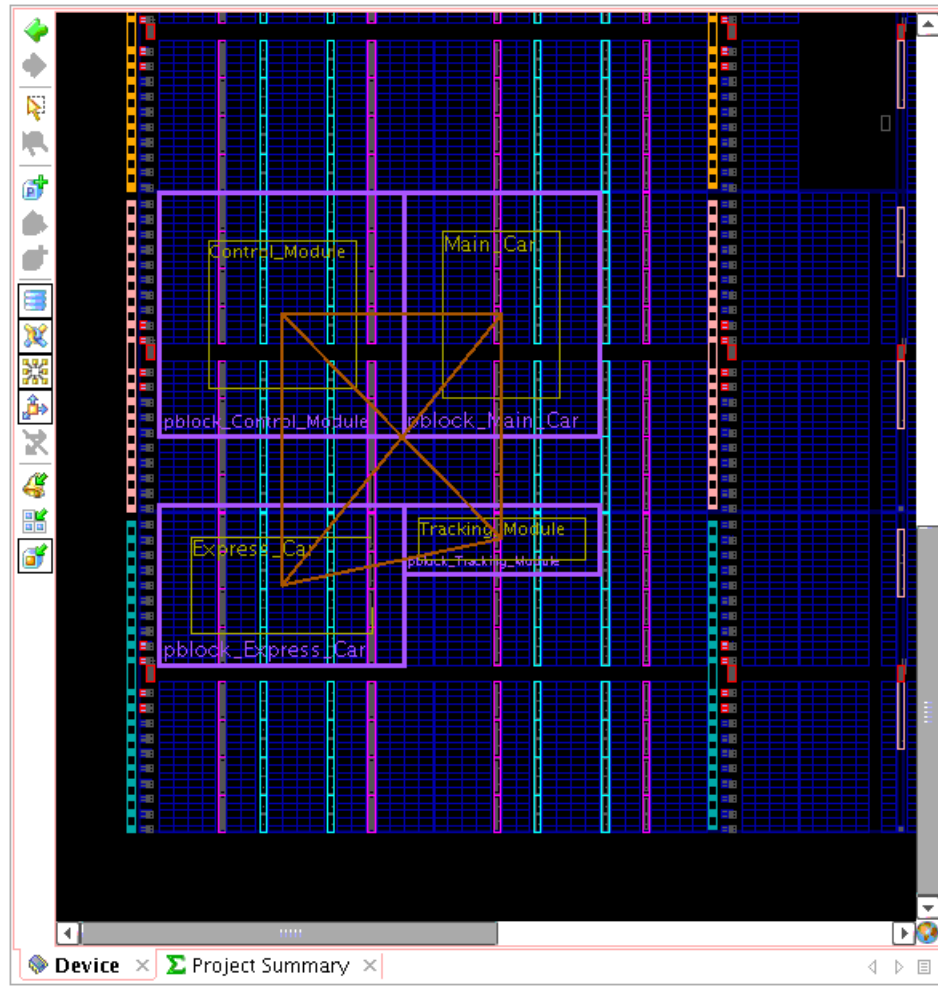


Figure 12: Setting Partitions in PlanAhead

Implementing a Partitioned Design

Now that partitions have been defined, and optionally floorplanned, the design can be implemented.

To implement the design, use the Implement button.

- For first time users and those interested in minimal setup requirements, click the green **Implement** button, which runs the implementation tools using the ISE default settings for NGDBuild, Map, PAR and TRCE. The implementation options can be modified prior to launching the run by clicking on the Implement button pull-down menu, selecting **Implementation Setting**, and making the necessary changes.
- For more advanced users, click the Implement button pull-down menu and select **Create Multiple Runs**, which can be used to create multiple implementations with varying implementation options.

The results of the implementation can be monitored in the Compilation Log window, from the status bar in the upper right corner of PlanAhead, or by opening the Design Runs view (**Window > Design Runs**).

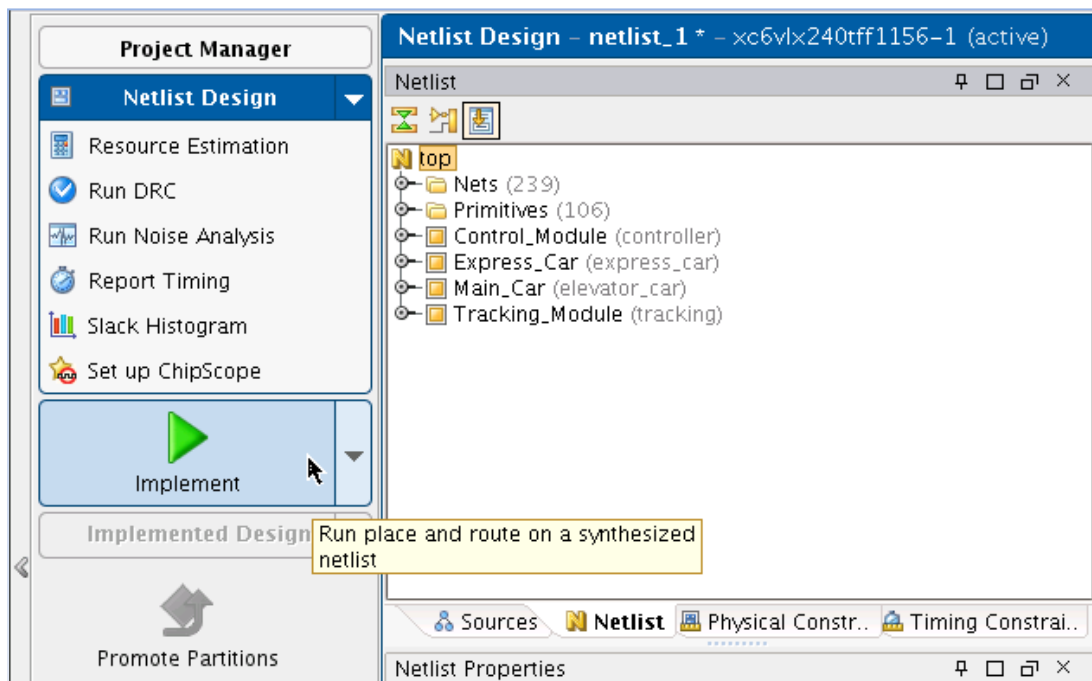


Figure 13: Using the Green Implement Button

Once the implementation has completed, the results can be loaded into PlanAhead by clicking the **Implement** button. You can verify that the partitions were properly defined and used by the implementation tools by viewing the implementation log files.

Promoting Partitions

When a partition has been successfully implemented, it must be promoted for future import. Promoting a partition is the same concept as exporting in the command line flow. To promote partitions from the active run, click the **Promote Partitions** button, as shown in Figure 14.

Note: Failure to promote a run will result in loss of the current partition results if the run is reset.

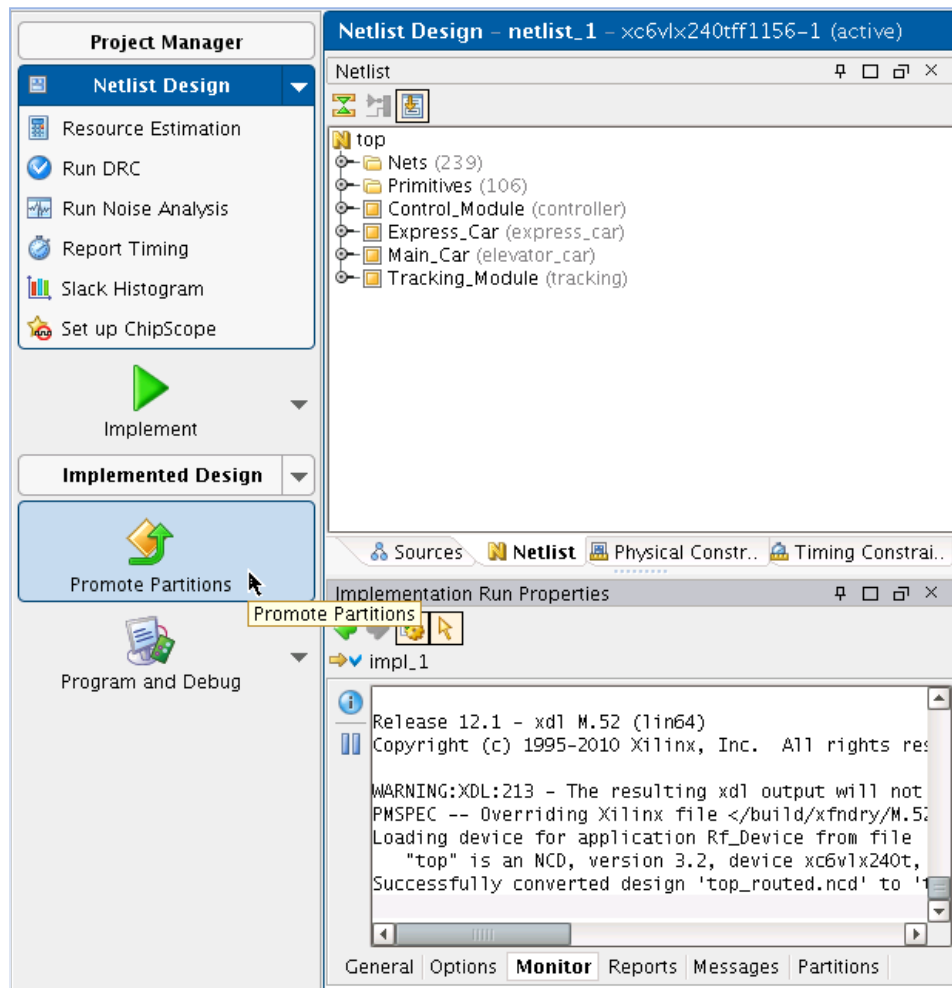


Figure 14: Promoting Partitions

Once the results have been promoted, the current run can be reset and rerun as many times as necessary. The promoted results will always be imported as long as the partition has not changed. Promoting new results will overwrite the current promoted data for a particular run. For more information on managing multiple runs and promoted data, refer to the *PlanAhead Tutorial: Leveraging Design Preservation for Predictable Results* (UG747) available on the Xilinx website:

http://www.xilinx.com/support/documentation/dt_planahead_planahead12-1_tutorials.htm

Managing Partition States

By default, the top-level partition is not selected for promoting. However, the top-level partition can be selected and promoted just like any other partition. All partitions that are promoted will have their state automatically updated to 'import' for the next run. Also, a partition that is not promoted can still be imported, but PlanAhead will not manage these partitions automatically. This can be managed manually in PlanAhead through the Partitions tab on the Implementation Run Properties window, as shown in Figure 15.

If a promoted partition gets out of date by updating a source netlist or modifying a physical constraint, the partition state must be changed to 'implement'. This change of state is not done automatically by PlanAhead. Failure to do so will result in the following NGDBuild error:

```
ERROR:NgdBuild:1037 - The logic for imported Partition '/top/Express_Car'
using previous implementation
'../../../../project_1.promote/Ximpl_1/top_prev_built.ngd'
has been modified. This situation can occur when the source for the
Partition was modified but the Partition was not re-implemented
and exported.
You must re-implement and export the Partition before it can be imported
into this design.
```

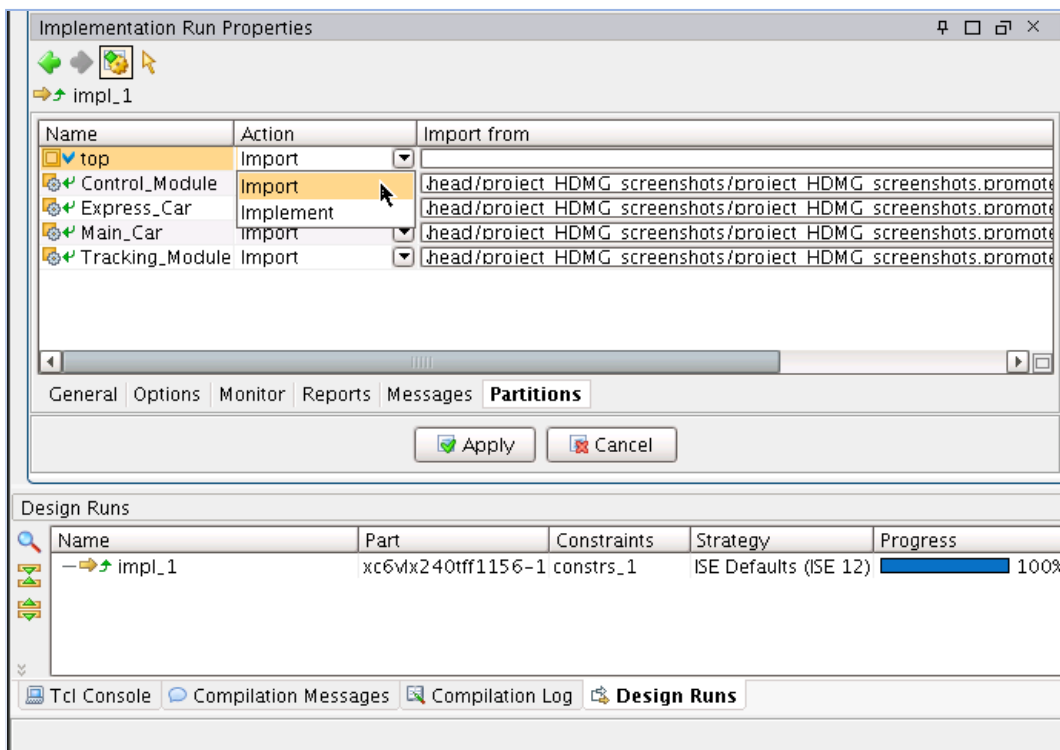


Figure 15: Managing Partition State in the Implementation Run Properties Window

Managing Design Runs

Once any necessary design changes have been made and partition states have been set correctly, the design can be re-implemented. The recommended flow for design preservation in PlanAhead is to maintain a single design run (impl_1), and continue to promote and re-implement as necessary.

For the example shown in this document, the design can now be re-implemented by clicking the **Implement** button once again. This will reset the run data and launch the implementation tools, importing partitions as defined by the Partition tab. Once this implementation completes, the results can be promoted, and previously promoted results are overwritten. As a result, only one promote directory contains the results for the design partitions. A single promote directory is easy to maintain, and avoids any complications caused by importing from multiple locations, as highlighted in the “Import Location” section in Chapter 1.

Debugging Partitions

Implementation Errors

Error When Trying to Import

If you get the following error, most likely the `xpartition.xml` file was not copied to the import directory.

```
ERROR:HierarchicalDesignC:154 - Import project "../import" is not an XPartition project.  
ERROR:HierarchicalDesignC:143 - Error importing Partition "/top/modulea".  
ERROR:HierarchicalDesignC:142 - Error found in XPartition file data.
```

What To Do If a Design Does Not Meet Timing When Importing

If a partitioned portion of a design does not meet timing, it might be helpful to relax the preservation level of the imported partitions to synthesis. This will allow small changes in placement and routing to the imported partitions. Since this gives the tools more flexibility, timing may be met. Once timing is met the partition should be exported, and the preservation level can be moved to 'placement' or 'routing' if desired.

If the design still does not meet timing, it may be necessary to change the state from import to implement on one or more of the partitions or to remove one or more partitions that span critical paths.

Also, standard flat flow timing closure techniques apply.

What To Do If a Design Does Not Place

The more partitions there are in a design and the higher percentage of logic that is imported, the fewer resources are available for the logic being implemented. At times, this can lead to a design that cannot be placed.

There is a message during place that can point to the issue.

```
WARNING: Place:1178 - 4 sites were unavailable during the placement phase
because they were already used by routing within preserved Partitions. If the
Placer is not able to find a successful solution, the following suggestions may
help the Placer find a solution.
```

```
1) To allow the Placer to use these sites, back off the preservation level
on one or more of the following Partitions. The following Partitions contain
unavailable sites:
```

```
Partition /top/Control_Module: 3 (preserve=routing)
Partition /top/Express_Car: 1 (preserve=routing)
```

```
If the problem persists, then careful floorplanning can minimize the number of
sites occupied by routing.
```

These sites are not usable because the imported routing makes one or more pins on an otherwise empty component unusable. Logic can be placed in these components if the router has the ability to move imported routing. This can be done by changing the preservation level to *placement* or *synthesis* on partitions with a high number of unavailable sites. If this is not enough a partition may need to be re-implemented.

Setting the top partition state to implement may free up enough resources so the placer can find a valid placement. If this is not enough, other partitions may have to be changed to implement also. The PlanAhead™ software can be helpful in the analysis of what partitions to change to implement by loading in the placement and routing information to the Results Viewer and highlighting each partition to see where its logic is located in the FPGA.

Also, too many partitions can have a negative effect on the utilization of a device. Removing partitions on modules that are not on critical portions of the design can help free up resources for other logic, and gives the tools more flexibility to find a solution that meets timing.

Careful floorplanning can also help designs find a final placement solution. If all the logic is allowed to spread out throughout the entire chip, it can be difficult for the placer to find resources for new logic that is being implemented. Floorplanning contains each partition to its own section of the design. Keep in mind that if an AREA_GROUP range is added or modified, the exported data of the partition is now out of date and the state of the partition must be set to implement for the next run.

What To Do If a Design Does Not Route

If a design with imported partitions does not route, the easiest solution is the change the preservation level to *placement* and re-run PAR. Map does not need to be re-run.

For the same reasons that floorplanning can help solve placement issues, it can also help solve routing issues.

If changing the preservation level does not work, and if the floorplanning has not helped, analysis of the placed design can help determine which partition(s) need to have the state changed. PlanAhead can be used to view the placement by loading in the placement and routing information to the Results Viewer and highlighting each partition to see where its logic is located in the FPGA.

What To Do If Slice Utilization Goes Up With Partitions

Some degree of increased utilization is expected due to optimization and packing restrictions imposed by partitions. However, this should normally be only a few percent and ideally negligible.

In certain cases utilization can increase by significant amounts. One cause for this is if ratio of LUTs to FFs highly favors one over the other. In a non-partitioned design the packer could combine some components from moduleA with components in moduleB. However, due to packing restrictions with partitions, components from partitionA cannot be packed with components from partitionB. This means empty slices must be used for this out-of-balance logic, causing slice utilization to increase.

To check whether a design is affected, look at the Area Group and Partition Summary section of the Map report (.mrp). A sample of this is shown below.

Partition "/top/moduleA":	
State=implement	
Slice Logic Utilization:	
Number of Slice Registers:	4,318 (4,318)
Number of Slice LUTs:	4,337 (4,337)
Number used as logic:	3,717 (3,717)
Number used as Memory:	620 (620)
Slice Logic Distribution:	
Number of occupied Slices:	1,750 (1,750)
Number of LUT Flip Flop pairs used:	5,249 (5,249)
Number with an unused Flip Flop:	931 out of 5,249 17%
Number with an unused LUT:	1,508 out of 5,249 28%
Number of fully used LUT-FF pairs:	2,810 out of 5,249 53%
Number of Block RAM/FIFOs:	3 (3)
Number of BUFDS:	1 (1)
Number of BUFR:	1 (1)
Number of GTX_DUAL:	2 (2)
Number of PLL_ADV:	1 (1)

Figure 16: Sample Map Report - Partition Resource Summary

There are two numbers for the utilization listed here. The first number is the resources for the individual partition. The second value which is listed in “()” is for the partition and all of its child partitions. This is stated at the top of the partition summary section in the MRP report, which can be overlooked. The section looks like the following:

```
Partition Resource Summary:
```

```
-----
```

```
Resources are reported for each Partition followed in parenthesis by resources
for the Partition plus all of its descendents.
```

BitGen DRC Errors

In certain cases a design will complete PAR, and then fail in BitGen with DRC errors like the following.

```
ERROR:PhysDesignRules:10 - The network <signal_OBUF> is completely unrouted
```

This is caused by an un-driven output of a partition connected to logic in the parent partition. Map cannot optimize the logic in the parent partition like it can in a flat flow, and the result is partially routed signals in the final design, which causes BitGen DRC errors.

This problem must be fixed at the HDL level, which will require any modified partitions to be implemented.

ChipScope Support

The ChipScope™ tool supports partitions but the user must change the partition state to 'implement' on any partition affected by a ChipScope core change.